Optimization and New Performance Evaluation of the Topic-Specific Trust Open Rating System

Holger Lewen

Institute AIFB, Universität Karlsruhe (TH), Germany
lewen@aifb.uni-karlsruhe.de

Abstract. In this technical report we describe the optimization of our implementation of the Topic-Specific Open Rating System (TS-ORS). We rerun our performance experiment and compare them to the old version of the code. Furthermore, we analyze the complexity of the algorithms used in the system.

1 Introduction

Taking the source and performance measures published in [1], we have reengineered large parts of the code to make the application run faster and with less memory consumption. In this report we will describe which steps were taken, how the architecture has evolved, and also how the performance has improved with the new code base. Also a few new features are described. The new code base will also be used in our Cupboard system [2].

2 Performance Limiting Factors in the Old Code and Reengineering

After profiling the old code, the main limiting factors were identified as database access and the implementation of multithreading. As a first step, the database connection pool was replaced by a faster one, which yielded in some speed-up, but was not sufficiently satisfying. The main speedup occurred when we started bundling the database write access, i.e., caching the statements inserting data into the database and then writing a huge insert instead of a couple of thousand small insert statements. Because this way less connections have to be requested form the connection pool, performance increases notably.

In terms of main memory consumption, most main memory is consumed by the big matrices we use for trust computation. The former approach for parallelization had multiple instances of a thread performing the same computations on different data. Taken one computation as a baseline, each additional computation run in parallel takes up the same memory. Running four threads thus means quadrupling the memory needed. We have changed the computations to now parallelize the computations by distributing parts of the computation over more threads. Whereas formerly the matrix multiplication step would have run

on one thread—but with more threads performing the same multiplication on different data in parallel—we now run one matrix multiplication, but distribute it over more threads. Since all threads can work on the same matrix, there is no increase in memory consumption. Whenever possible, we reengineered the core methods of the algorithm to support multithreading.

For the meta-trust propagation, we redesigned the algorithm so that the propagation is done directly in the database, without many reads and writes from the java program. This also caused a significant speed-up.

2.1 Changes in Architecture

We have updated both the database schema, and the structure of the source code.

Database Schema While most of the schema has stayed the same, we have introduced a new table *localnotrust* where information is stored about which users are not connected to the Web of Trust (WOT). This table is then used for a faster information lookup at runtime. The updated schema can be found in Fig. 1. We have also updated the index structure to improve performance. As before, when the re-computation is triggered during runtime, temp versions of the tables *runtimetemp*, *localtrust*, *globaltrust*, and *localnotrust* are created. When the computations are completed, they replace the tables used at runtime.

2.2 UML-Diagram

Because of the performance optimization explained above, also the design of a number of classes and methods has changed. The updated UML Class-Diagrams can be found in Fig. 2 and Fig. 3. We will now lay out the changes and new functionality.

Changes to TS-ORS Core One of the most important changes is the redesign of the multithreading functionality. The class *Multithreading* was removed, and the Jama *Matrix* class extended by methods that allow multithreading. The new *Caching* class allows to cache results based on standard parameters. The *computations* class was cleaned up by moving small methods into bigger ones, and extended with a method to retrieve the trust statistics for a review.

2.3 Interaction with other Programs

The number of Java Servlets used for the integration with Cupboard has been increased, to offer access to the new functionality of retrieving the trust statistics for a review, and also to trigger the caching mechanism. The new UML class diagram can be seen in Fig. 4.

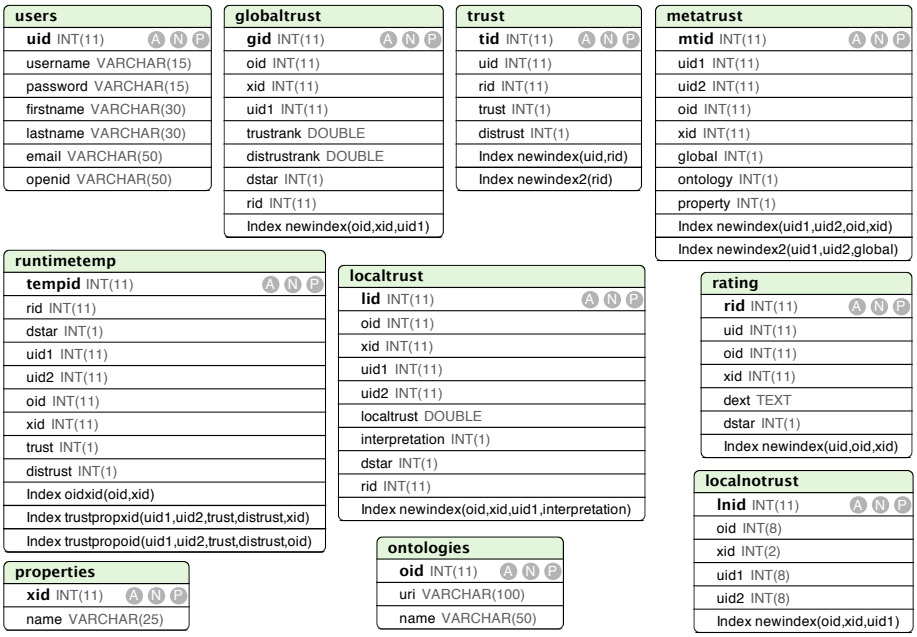

Fig. 1. This graphic depicts the new database schema of the TS-ORS.

3 Complexity Analysis

In order to understand the behavior examined during the benchmarks, it is also important to analyze the complexity of the algorithms. We have to distinguish between the computations run offline (the trust computation), and the computations needed at runtime (ranking of reviews, computing an overall score for an ontology). We start by analyzing the computations performed offline. For this we use the common Big-O notation [3].

3.1 Complexity of Trust Computation

Since the Big-O notation is dominated by the part of the algorithm having the highest complexity, we will analyze each step taken separately to determine the Big-O complexity. Since constants are irrelevant for the complexity analysis, we do not try to provide the exact number of times an operation is performed, but concentrate on the complexity of the operation performed. Usually one input parameter is chosen for the complexity analysis, since the result is easier to understand than trying to combine all variables.

In our system, we have a number of variables. The number of ontologies, the number of properties, the number of users, the number of reviews, and the number of trust statements on these reviews. We will consider the number of users as our input parameter, since it is the one having the largest impact on

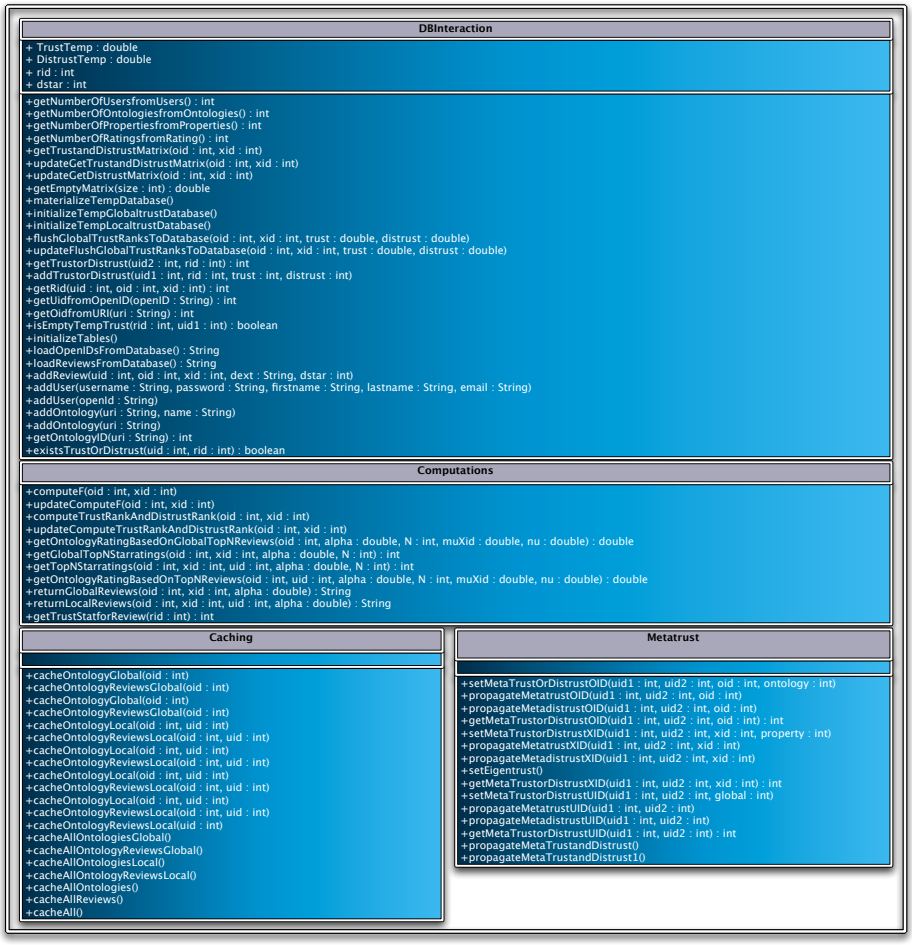

Fig. 2. This graphic depicts the updated UML class diagram of some of the TS-ORS core components.

Fig. 3. This graphic depicts the updated UML class diagram of the rest of the TS-ORS core components.

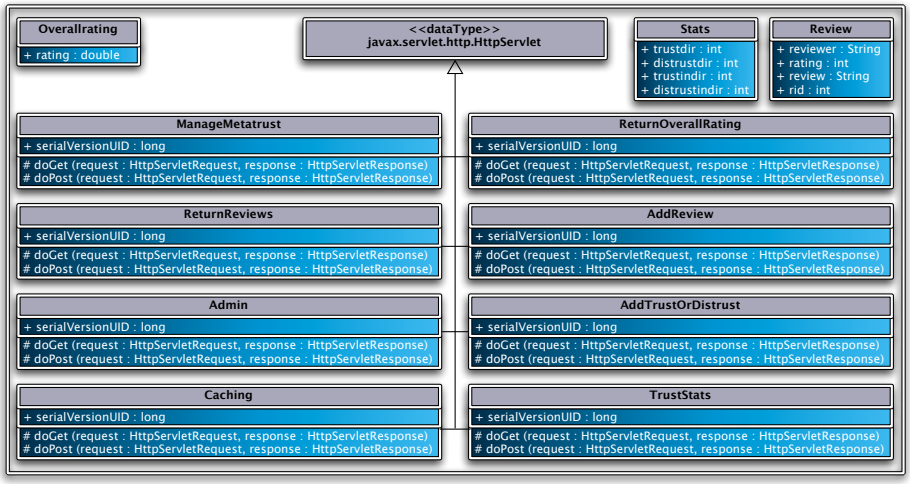

Fig. 4. This graphic depicts the new UML class diagram of the servlets used for interaction with Cupboard.

the complexity. It is important to understand that in the default setting the computations are performed exactly $\#ontologies * \#properties$ times (since the trust is computed for each ontology–property combination individually).

When the offline computations are triggered, the following operations are used:

- Matrix addition (in $O(n^2)$)
- Matrix subtraction (in $O(n^2)$)
- Matrix transponition (in $O(n^2)$)
- Matrix multiplication (in $O(n^3)$)[1]
- Matrix scalar multiplication (in $O(n^2)$)
- The majority rounding (in $O(n^3)$) which consists of n times the following:
  - Sorting (in $O(n * \log n)$)
  - Interpreting Trust (in $O(n^2)$
- TrustRank computation (in $O(n^2)$)
- DistrustRank computation (in $O(n^2)$

Since the performed database operations all have a complexity below $O(n^3)$, the overall complexity of the computations is in $O(n^3)$. As long as there are not too many users, there is no problem computing everything for every ontology and property separately. If the time taken to compute the trust and distrust is taking too long for the needs of the application, the number of times the computation is performed can be reduced by grouping the available trust information and only perform the computations where they promise the most gain. Possible groupings could for example be by property or by ontologies of a special area.

[1] We are aware that other methods for matrix multiplication exist that have a lower complexity, but their overhead is too big to make them useful in our case

3.2 Runtime Complexity

Here we have to distinguish two tasks the TS-ORS performs at runtime: Ranking the reviews for a given ontology–property combination, and computing an overall rating for an ontology. We also have to distinguish the cases that the user is identified (and local trust can be used), and the case where the user is unknown (and global trust has to be used).

Ranking of Reviews For this task the system is given as input parameters the ontology and property for which the reviews have to be retrieved, and the user who is requesting the information (if available). Also the combination of trust and distrust can be influenced by a parameter. For the result based on the global reviews, the method basically just performs one database query and then provides the ordered reviews as output. Its runtime is dependent on the number of reviews that exist for this ontology-property combination. Within the database query, the results are ordered. So with n being the number of reviews, the complexity of the retrieval is in $O(n*\log n)$ which is mainly due to the sorting needed. In case the user is known, more database queries are performed, but the complexity stays in $O(n * \log n)$ with n being the number of existing reviews. Since there cannot be more than 1 review per user for each ontology–property combination, the number of users is an upper bound for the number of reviews. Most of the times, the number of reviews will be far smaller than the number of users.

Overall Rating of Ontologies In case only the overall rating of an ontology has to be computed, the complexity is the same as for the ranking of the reviews, since for all properties of an ontology, the top n reviews have to be retrieved. Since during this process the results are sorted in the database, the complexity is $O(n*\log n)$ for both retrieval based on global and on local trust, with n being the number of reviews. Since the number of properties is a constant factor in any installation of the system, it does not influence the complexity. For runtime performance it is important to know that limiting the number of top reviews based on which the overall rating is computed does lower the computation time, but does not affect the theoretical complexity. Again, the number of users is an upper bound for n.

3.3 Further Optimization

While it is not possible to lower the worst case complexity, we have tested successfully an optimization for scenarios, where many users are not connected to the web of trust for a given ontology–property combination. Imagine the case where there are a million users in the system, but only a few reviews and trust statements on these reviews. Running the computation with a million times a million user matrix would result in a serious performance problem. Luckily, we managed to address this case by reducing the number of users taken into account

for the computation to these actually affecting the outcome of the computation. By the nature of the algorithms, the only users actually affecting the outcome of the computation are the users who wrote the reviews for that ontology–property combination, and the users that have made a (dis)trust (or meta-(dis)trust) statement covering one of these reviews. In order to only use the data necessary, we first get the IDs of all users that have either written or (dis)trusted reviews, and fill two hash maps with a mapping from original ID to new ID and vice versa. The new IDs start with 1 and are auto incremented. The trust and distrust matrix are then filled with the data using the new IDs, and after the computation is done, data is written back using the old IDs. The result is exactly the same as if the algorithms were run on the complete user-base. The overhead is very small, since writing and reading from the Hash table can be done in almost constant time. If we use this technique, then we can abandon the *localnotrust* table, since we would not have an entry for all the users that we did not consider for the computation. In our algorithm at runtime, we derive these users instead by first seeing which authors wrote reviews for that ontology–property combination, an who of them is not trusted by our user under consideration. This also saves space in the database.

Using the optimization, we can ensure that the complexity is in $O(m^3)$, with m being the number of users who wrote or (dis)trusted reviews. In the worst case, $m = n$, but in the normal data-sparse setting, we can save time during the computation.

4 Evaluation

In order to make the benchmarking results comparable to the one of the benchmark, we have left the evaluation methods unchanged and have just run it using the new code. So the following diagrams will be the same as in the older deliverable, just with updated times. The Setup is exactly as described in [1]. In order to compute a more accurate average time, we have run the overall computation task 500 times instead of 300 times. We did not use the optimization described in Section 3.3, because our scenarios were not data sparse.

4.1 Results

We have performed the benchmark again on two systems: A Dual-core MacBook Pro running Mac OS X 10.5.8 and a QuadCore PC this time running Ubuntu 64bit.

In order to find out how a different number of reviews for each item and trust statements on these reviews would influence the computation time, we did not only vary the number of users (100, 250, 500), but also the amount of reviews and trust statements available, resulting in 9 different test runs. During the generation of the test data, for each different user group size, we had one setting which had 10& of the users review each ontology–property combination and 10% of the users then trusting or distrusting each of these reviews, one with

a 50%–50% distribution and a worst case scenario (100%–100%). Worst case means that every users reviews every ontology–property combination, and also every user then votes on the usefulness of these reviews. In a realistic setting, the distribution is likely less than 10%–10%. We have furthermore assumed that of the users making trust-statements, 70% of the trust statements were trust, and 30% were distrust. For the meta-trust generation, we have fixed the percentage of global, ontology- and property-specific meta-trust statements to 20% per user for all runs, i.e. each users meta-trusts 20% of the other users. The test data was regenerated between all runs, to prevent caching effects in between runs. As for the number of properties, they were fixed to 5. Furthermore we limited the number of ontologies to 12. As shown in section 3 the complexity of the computation does not increase by having more ontologies, since the ontologies are just a linear factor (it will take 10 times longer to perform the computations for 120 instead of 12 ontologies). This is because all the computations have to be performed for each ontology–property combinations.

MacBook Pro The computation was performed on a MacBook Pro with 2.16 GHz Intel Core 2 Duo Processor and 3GB 667 MHz DDR2 SDRAM running Mac OS X 10.5.8. The hard disk is a ST9320421ASG (320GB 7200 RPM, 16MB Cache, avg. seek time 11 ms). This time we decided to use Java 6 64bit with Eclipse Version is 3.5 for Mac. MySQL versions are 5.1.36 64bit for the server and 5.1.8 for the J-Connector.

The results are shown in the figures 5, 6, 7, 8 and 9. The percentages in the labels refer to the percentage of reviews, trust and meta-trust statements generated as test data (see explanation above). For example 10%10%20% means that 10% of all users have reviewed each ontology, 10% of all users have rated each review, and each user expresses meta-trust towards 20% of the other users. We believe that this setup allows to draw some conclusions about the scalability of the system with regard to number of users and sparsity of data. In the following section we will analyze the results seen in the figures presented here.

QuadCore Vista 64bit The computation was performed on a 2.40 GHz Intel Core 2 Quad Q6600 processor with 8GB 800 MHz DDR2 SDRAM running Ubuntu Kernel 2.6.28-15 x86_64. The hard disk is a Samsung HD103UJ (1TB, 7200 RPM, 32MB Cache, avg. seek time 8.9). Eclipse Version is 3.5.0 for Linux 64bit. Java version was OpenJDK Runtime Environment (IcedTea6 1.4.1) 6b14-1.4.1-0ubuntu11. We used MySQL 5.4.1 beta linux x86_64 ICC-GLIBC23 for the server and 5.1.8 for the J-Connector.

5 Result Analysis

We will now analyze the improved results of the benchmark and try to explain how this speedup was gained. The general analysis provided in the last deliverable does still hold. We will start with figures 5 and 10, which depict the

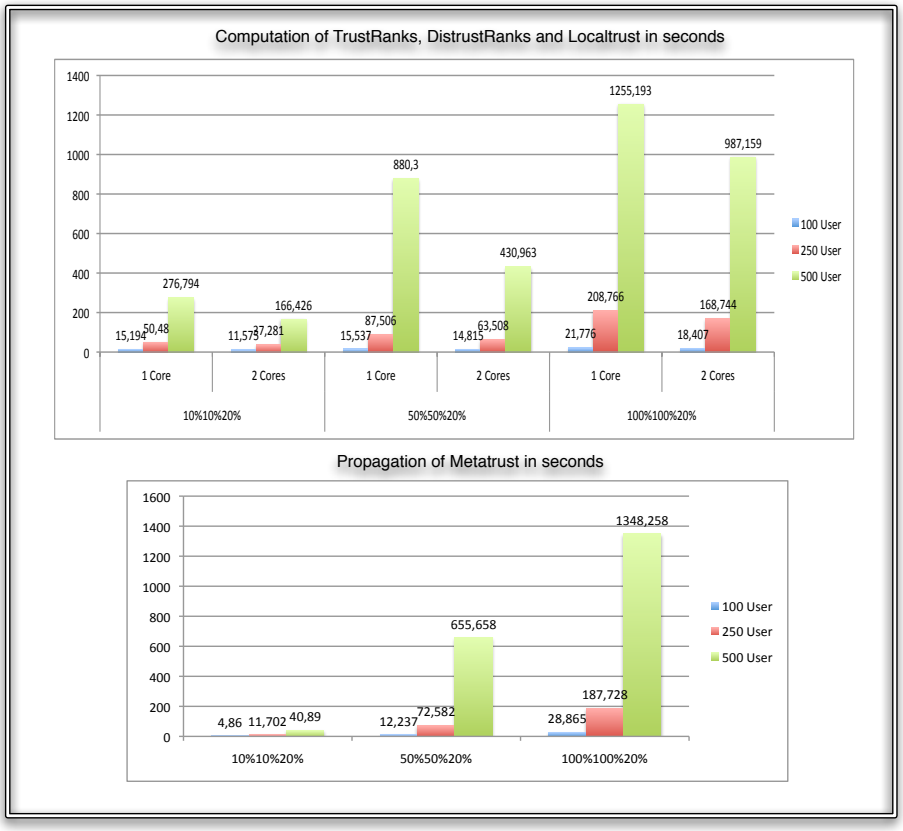

Fig. 5. This graphic depicts the results of the benchmark of the computation and meta-trust propagation. Data is presented as time in seconds. (Run on MacBook Pro)

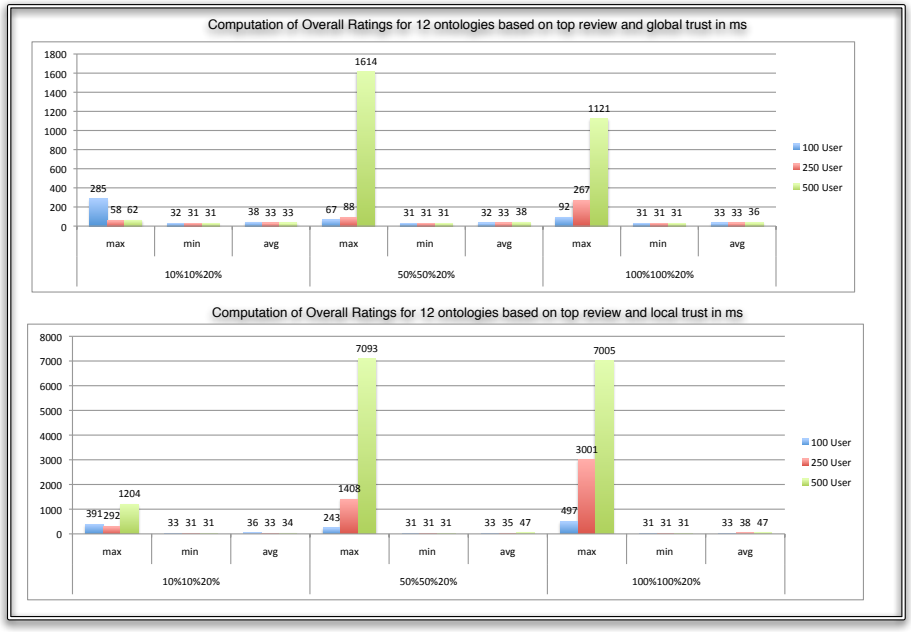

Fig. 6. This graphic presents the times taken (min, max and avg.) for returning the overall rating for all 12 ontologies in ms once for global and once for local trust. 500 runs were performed and only the top review was considered for the computation. (Run on MacBook Pro)

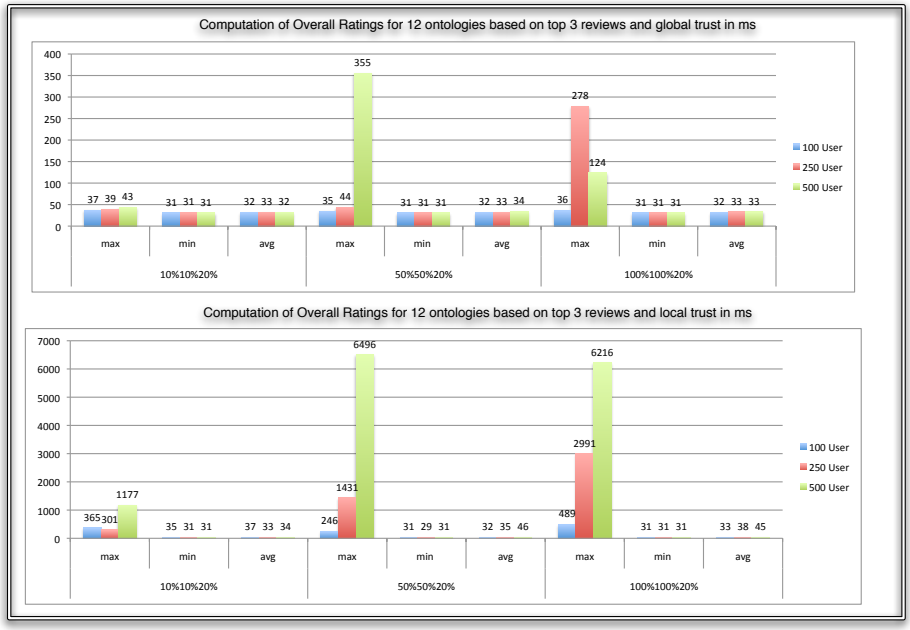

Fig. 7. This graphic presents the times taken (min, max and avg.) for returning the overall rating for all 12 ontologies in ms once for global and once for local trust. 500 runs were performed and only the top 3 reviews was considered for the computation. (Run on MacBook Pro)

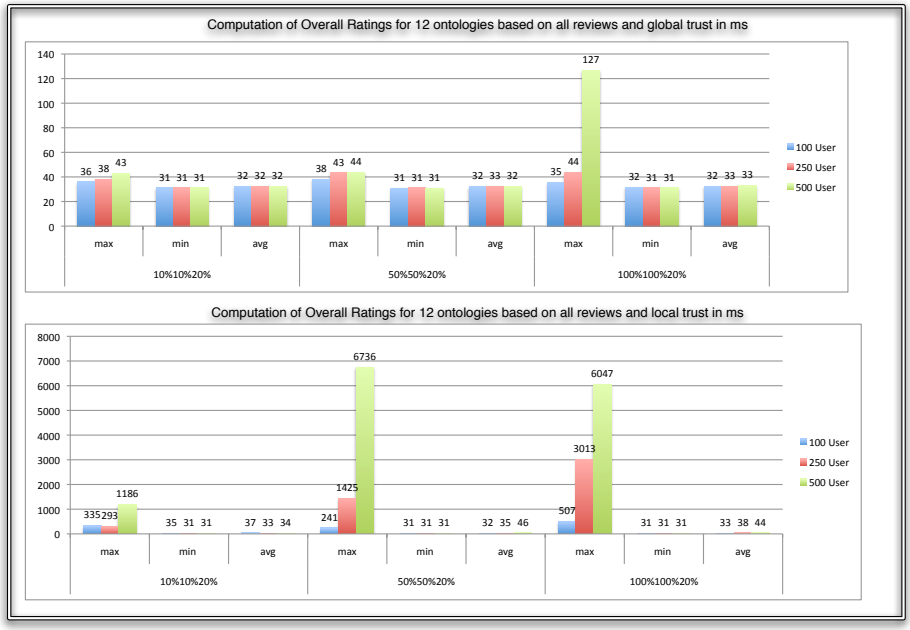

Fig. 8. This graphic presents the times taken (min, max and avg.) for returning the overall rating for all 12 ontologies in ms once for global and once for local trust. 500 runs were performed and all reviews were considered for the computation. (Run on MacBook Pro)

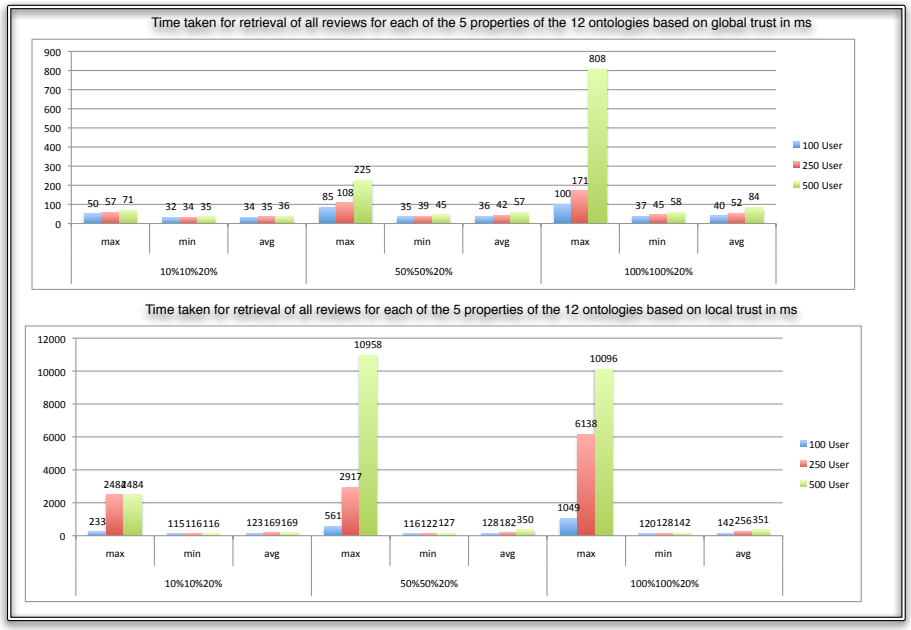

Fig. 9. This graphic presents the times taken (min, max and avg.) for returning all reviews for all 5 properties of all 12 ontologies in ms once based on global and once based on local trust. 50 runs were performed. (Run on MacBook Pro)

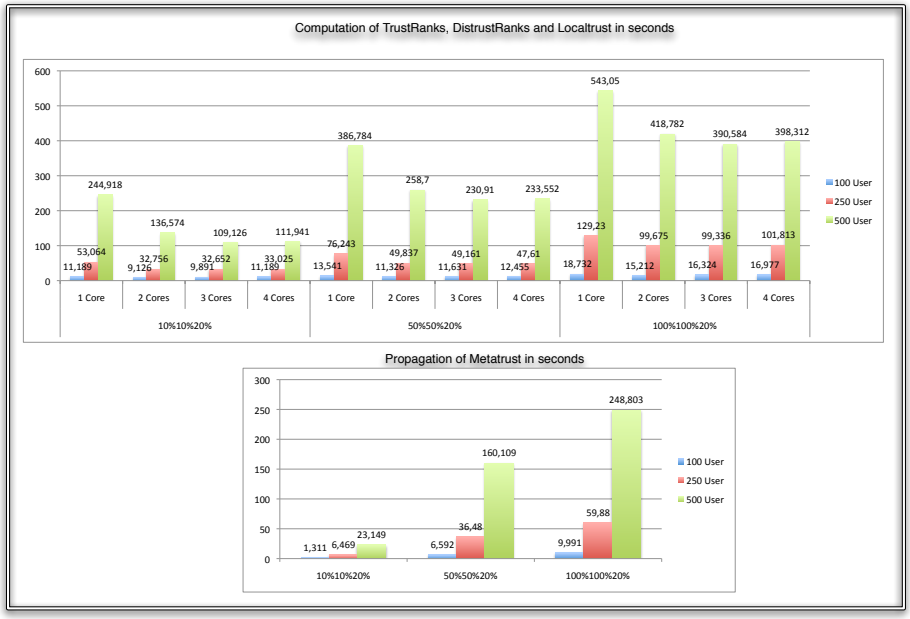

Fig. 10. This graphic depicts the results of the benchmark of the computation and meta-trust propagation. Data is presented as time in seconds. (Run on Intel Core 2 Quad)

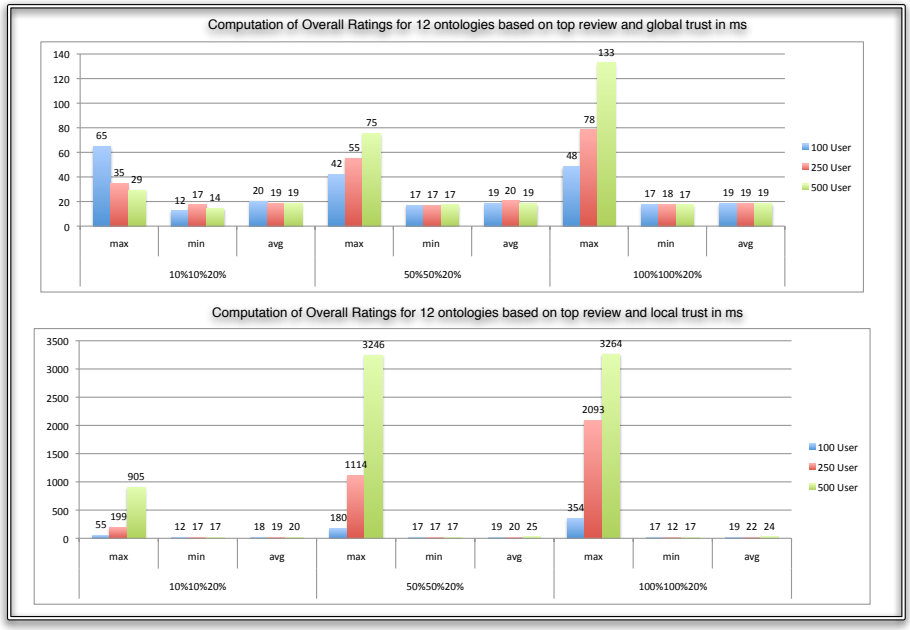

Fig. 11. This graphic presents the times taken (min, max and avg.) for returning the overall rating for all 12 ontologies in ms once for global and once for local trust. 500 runs were performed and only the top review was considered for the computation. (Run on Intel Core 2 Quad)

Fig. 12. This graphic presents the times taken (min, max and avg.) for returning the overall rating for all 12 ontologies in ms once for global and once for local trust. 500 runs were performed and only the top 3 reviews was considered for the computation. (Run on Intel Core 2 Quad)

Fig. 13. This graphic presents the times taken (min, max and avg.) for returning the overall rating for all 12 ontologies in ms once for global and once for local trust. 500 runs were performed and all reviews were considered for the computation. (Run on Intel Core 2 Quad)

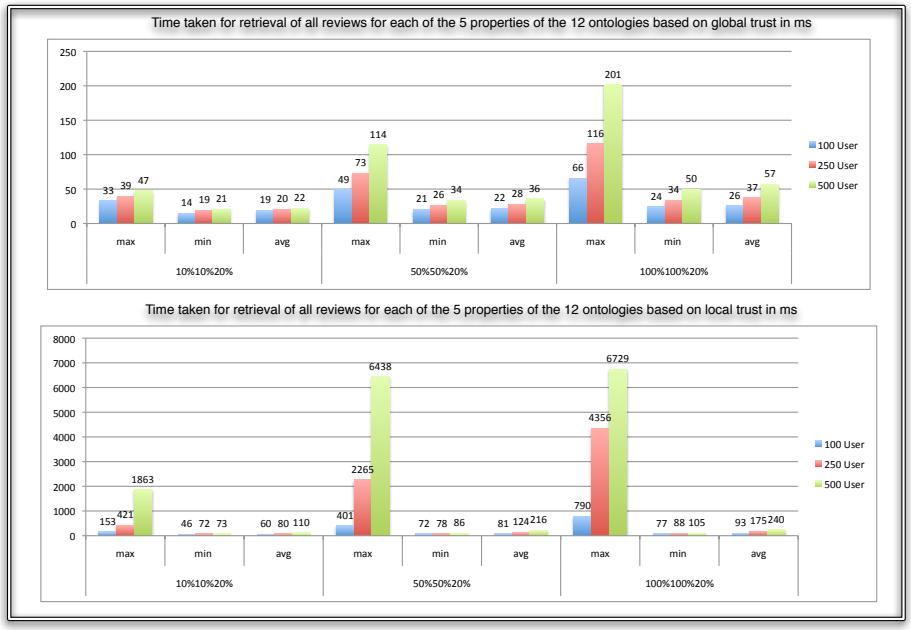

Fig. 14. This graphic presents the times taken (min, max and avg.) for returning all reviews for all 5 properties of all 12 ontologies in ms once based on global and once based on local trust. 50 runs were performed. (Run on Intel Core 2 Quad)

computation of trust and the propagation of meta-trust. In order to understand which steps are performed int these two events, we will now explain them in more detail.

5.1 Meta-Trust Propagation and Trust Computation

As can be seen better in Fig 15, which provides a direct comparison between both the old code and the new code on the two test computers, the performance for Meta-Trust propagation has increased on average by a factor of 10 (over all settings and only comparing the results on the same machine). The same is true for the performance of the trust computation. It can be seen that main memory is a limiting factor in the computation when looking at the comparison for the new code on the Core 2 Duo and Core 2 Quad. While for smaller scenario they perform more or less equally fast, for the scenario with more data (e.g.100%100%20% with 500 users), the difference between the two computers increases. This is because on the smaller machine with only 3 GB of Ram, not everything can be kept in main memory, and is stored on disk (Swap file). When this happens, the performance decreases, since disk access is much slower than main memory access.

Leaving memory related variations in the execution time aside, it seems that the duration for Meta-Trust propagation grows linearly with the increased amount of data, and roughly squared with respect to the number of users (also taking into account that the amount of data grows with the number of users). For example: The time taken for 100 users in the 100%100%20% setting on the Core 2 Quad is 10 seconds, for 5 times as much users it is 250 seconds ($= 10 * 5^2$). The time for 500 users in the 10%10%20% setting is 23 seconds compared to the 250 seconds in the 100%100%20% setting. It still has to be noted that this is mainly due to the bigger number of reviews and trust statements that we set to a fixed portion of the users. So for the 10%10%20% that means that in the case of 500 users, we have 5 times the amount of reviews, trust- and meta-trust statements. If we only increase the number of users, but not the number of reviews, the computation takes roughly the same time. For example, the propagation of meta-trust for the setting 500 users with a 2%, 2%, 4% setting (which is equivalent to the 100 User 10%10%20% setting) takes 4.5 seconds instead of 4.9 seconds for the case of 100 Users (which indicates that the number of users alone does not affect the duration of the Meta-Trust propagation, mainly the amount of data to process is decisive). The results for this comparison can be seen in figure 16.

The speedup for the computation was mainly due to the improved database interaction. The database access is sort of the lower barrier for the execution time which cannot be lowered any further. Looking at Fig. 17 it can be seen how the distribution of time changes when the computation is parallelized. Since database access is not parallelized, this time is solely depending on the amount of data inserted and read and stays the same even if more threads are used for the computation. The computation itself sees improvement, but here the overhead for parallelization has to be regarded, which explains why the time taken for

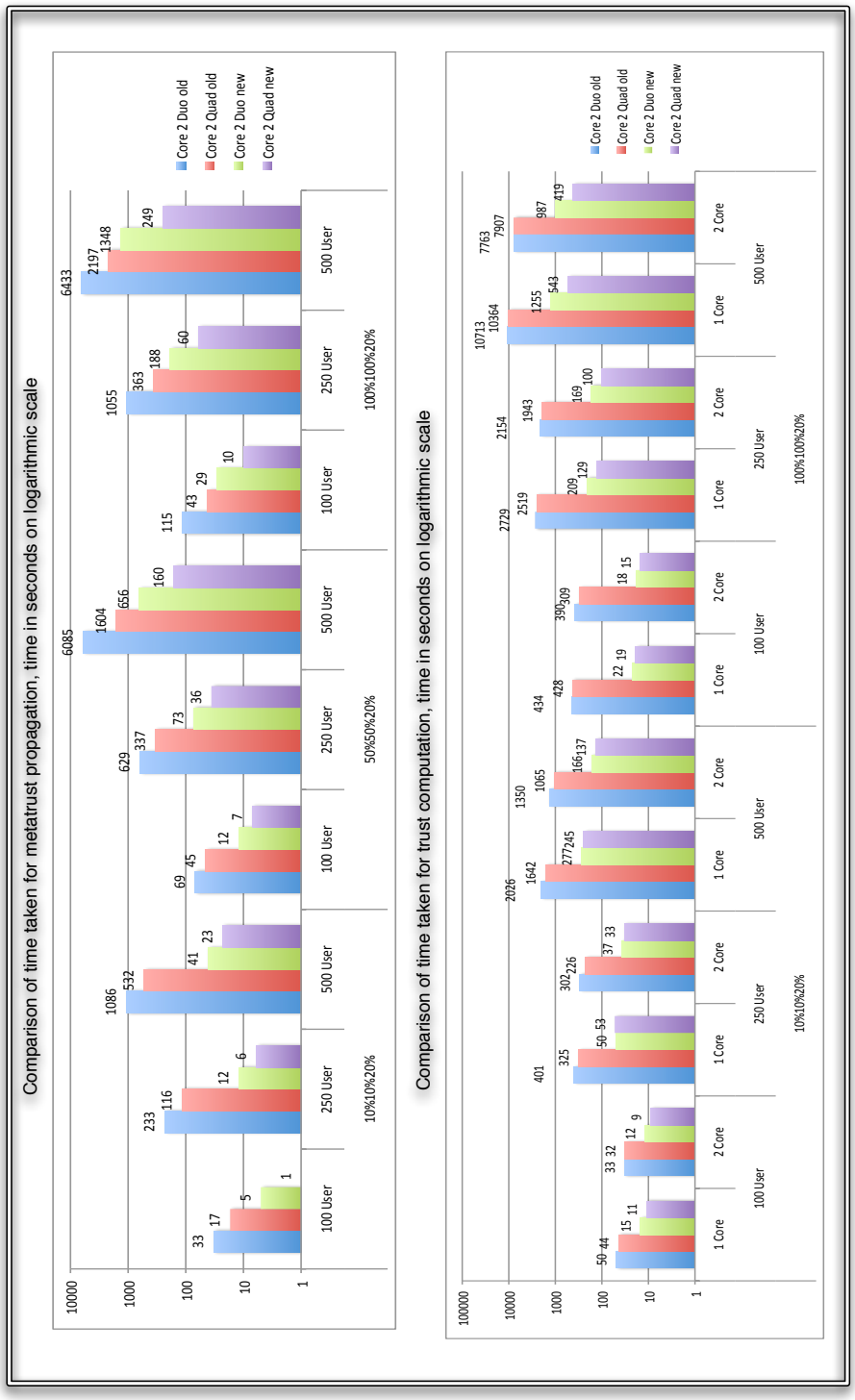

Fig. 15. This graphic presents a direct comparison between the two versions of the code and the two different computers. Please note that the left axis has a logarithmical scale.

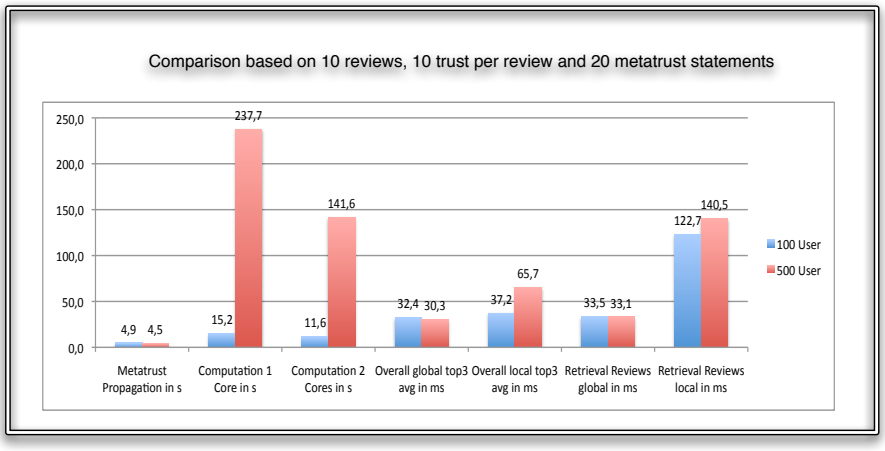

Fig. 16. This graphic presents a comparison between the time taken for different actions based on the same review and trust statements but increased number of users. (Run on MacBook Pro)

computation does not go down linearly with an increasing number of threads used. For most realistic scenarios, it seems at least 2 and at most n-1 threads should be started on a multi-core processor.

The results seen in the figures are for computing all 60 ontology–property combinations. So you have to divide the results by 60 to know how long one trust computation cycle lasts. For updates during runtime (a single trust statement was added), it is possible to recompute the trust on the fly, since with the new code, even in the worst case setting, the computation time is now around 6 seconds per ontology-property combination. In a more realistic setting (few reviews and trust statements) this time is now less than 1 seconds.

5.2 Overall Computation

In order to see how the different parameters influence the time for retrieving the overall rating for an ontology, we have based the computation on only the top review, the top 3 reviews and all reviews (just for worst-case considerations). For each of these settings, we base the computation on both local trust and global trust. Furthermore, we run each computation 500 times, to get more accurate average results. We measure the execution time for each of the 500 runs, and present the maximum time as well as average and minimum time needed for providing the result. It is intended to make use of the databases caching techniques, since they can also be exploited in real systems using the caching method (see section 2.2 above). For the local trust based computation, we changed the user for which the results were computed in between the 3 different settings (top-1, top-3, all) so that results would have to be re-cached). The results can be found in figures 6, 7, 8, 11, 12 and 13.

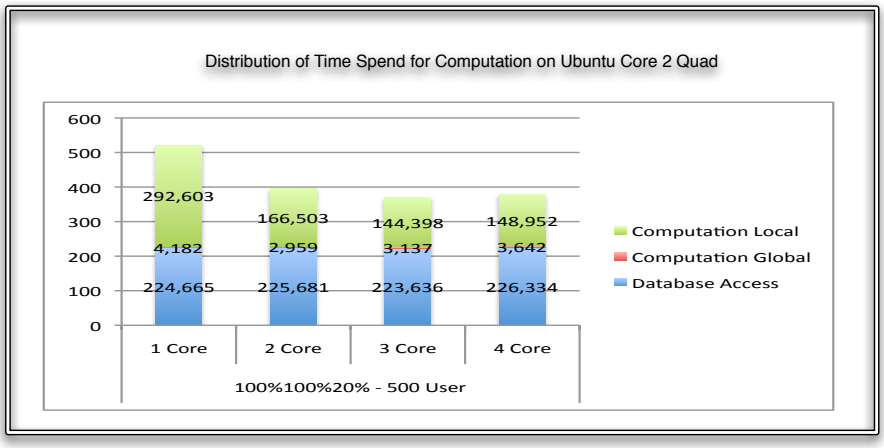

Fig. 17. This graphic presents the distribution of time spend for the different tasks during the trust computation for a different number of threads.

The overall computation is one of the most important features of the TS-ORS and it is performed constantly at runtime. So here a quick response time is far more important than for the larger computations which are performed offline and only at dedicated time-points.

The results indicate that while the maximal time in case of a cache miss or for other system-specific reasons can be in the order of seconds (bare in mind that this is for all 12 ontologies), the more significant average is relatively independent of the amount of users or the number of reviews in the system. With the new code, also the average retrieval time for global and local trust seems to be the roughly the same. We have compare the results of the old and the new code for in Fig. 18. Please note that we have chosen to compare the minimum times rather than the average times, because we have used a different number of runs during the two performance benchmarks, and the only comparable time is the minimum time. Also note that the more number of runs you use, the closer the average time gets to the minimum time. The maximum time is mainly encountered when there is no cached information (cache miss). For our best machine, the average minimum time to retrieve both local and global based results now is 17 ms for all 12 ontologies. That means that we have improved computational performance to roughly 1.5 ms in the best case.

5.3 Review Retrieval

When a user wants to browse reviews for ontology–property combinations, it is important to retrieve them in a personalized order. So when a user is logged in, the reviews are retrieved in an order based on local trust of this user, otherwise they are ordered according to global trust. We have benchmarked retrieving all of the reviews in aforementioned personalized order for all ontology-property

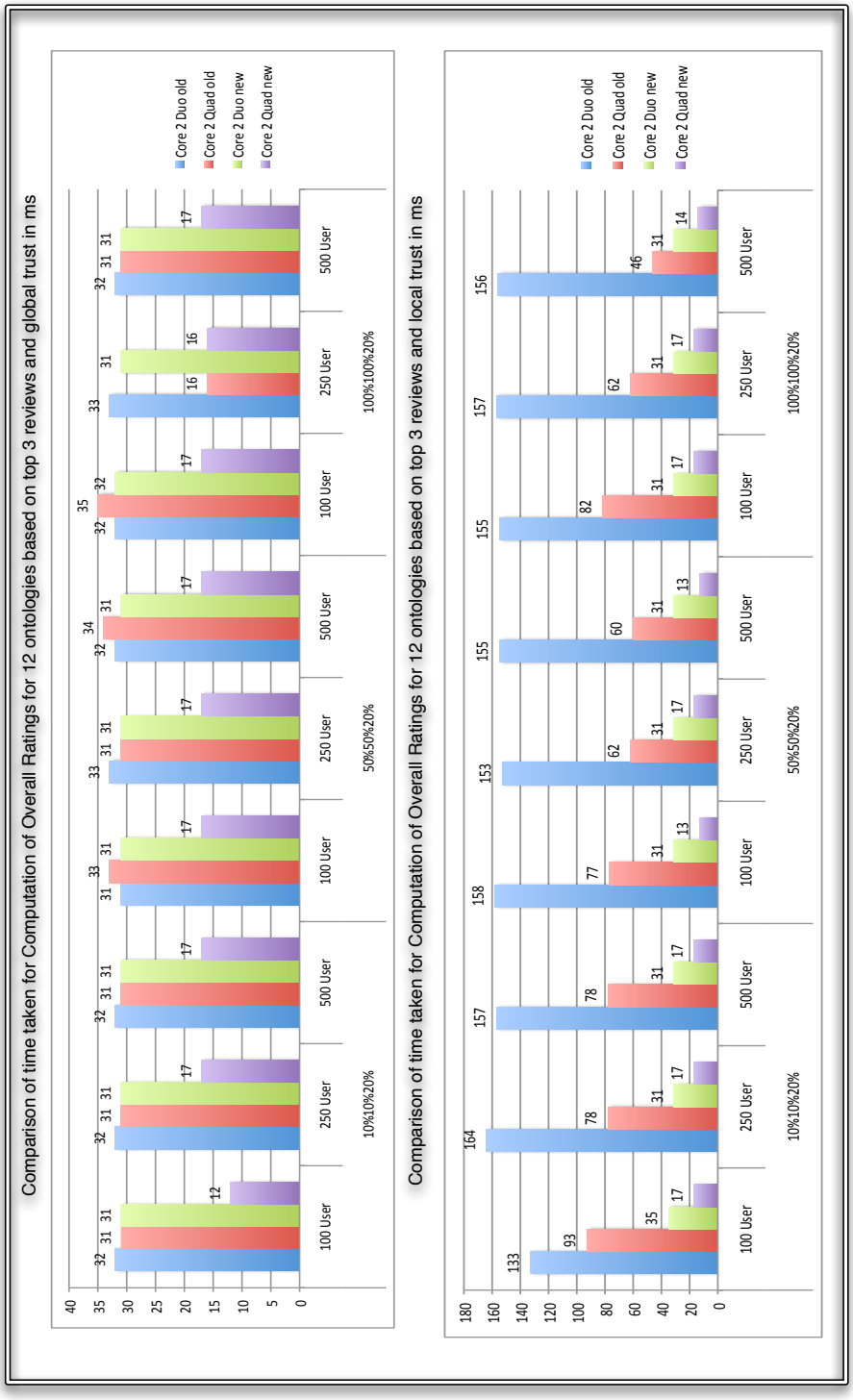

Fig. 18. This graphic presents a direct comparison between the two versions of the code and the two different computers. Please note that only min times are compares and the timer is in ms.

combinations. In figures 9 and 14, you can see the results both based on local trust and global trust. We have run each task 50 times to compute accurate results.

This task will also often be requested at runtime, when a user is browsing different ontologies. Since all the reviews have to be ordered, results are better when there are not too many reviews. For a realistic scenario (e.g. the 10%10%20%), the time to retrieve the ordered reviews is around 0.5 for each ontology–properties based on global trust (depending on number of users and thus reviews). Retrieving the results based on local trust takes about 1 ms on the fast machine.

6 Results Learnt from Benchmark

Since many operations rely on a fast database, having good hard disks and enough caching enabled for the database is key to obtaining a good performance. In order to minimize query time at runtime, after each computation of trust, the caching method can be called to cache results, for example for logged in users. This leads to fast runtime response times.

It also seems to be sensible to use at least a dual-core machine with 2 threads for the trust computation, since the results are much faster than those obtained for a one-core solution. Also sufficient memory and ideally a 64-bit system with 64-bit java improve the performance. Depending on how important it is to take the latest user data into account, the frequency of overall re-computation can be increased or lowered. Amazon.com, for example, take 24 hours to take a trusts statement into account. In case a really fast re-computation is needed, the computation can be distributed among different machines, each containing a database filled with only the necessary information. Since the computation of trust is independent for all ontology–property combinations, in the most extreme case, you could use one machine per combination and later merge the results.

The runtime performance looks even more promising after the code optimization, and should not lead to bottlenecks at runtime.

References

1. Lewen, H.: Implementation and performance evaluation of the topic-speciÞc trust open rating system. Technical report, UniversitŠt Karlsruhe (TH) (JUN 2009)
2. d'Aquin, M., Lewen, H.: Cupboard – a place to expose your ontologies to applications and the community. In Aroyo, L., Traverso, P., Ciravegna, F., Cimiano, P., Heath, T., Hyvšnen, E., Mizoguchi, R., Oren, E., Sabou, M., Simperl, E.P.B., eds.: The Semantic Web: Research and Applications, 6th European Semantic Web Conference, ESWC 2009, Heraklion, Crete, Greece, May 31-June 4, 2009, Proceedings. Volume 5554 of Lecture Notes in Computer Science., Springer (MAY 2009) 913–918
3. Knuth, D.E.: Big omicron and big omega and big theta. SIGACT News **8**(2) (1976) 18–24